# netEvo

*A Java package for the Evolution of Artificial Neural Networks*

August Mayer, January 2010
Helmut A. Mayer, February 2014

# Table of Contents

# 1. Overview

The netEvo package is a meta-package for the evolution (JEvolution package) of Artificial Neural Networks (Boone package) assisting specifically in ANN encoding and fitness evaluation. For proper usage of netEvo you should have a basic understanding of the underlying packages. The main functions of netEvo are:

- Evolution of ANN structure and parameters (e.g., weights)
- Evolution of training data sets
- Additional ANN training during evolution
- Various fitness functions with network regularization
- Modular encoding of features to be evolved
- Bit, integer, or real value encoding of features
- Grouping of features onto different chromosomes
- Evolution of activation functions
- Evolution of input/output neurons
- Evolution of Spiking Neural Network (SNN) parameters (e.g., thresholds)

# 2. A short introduction

The base classes of netEvo are situated in the **netevo.*** package:

- Evolver
- NetEvolver
- TrainEvolver
- NetPhenotype

The **Evolver** is the "main" class of netEvo, where the ANN template has to be defined. For ANN evolution the template network is the base network representing the "maximal" network, i.e., evolution will always prune neurons and links from the template structure (if the corresponding evolution features are set by the user). Hence, an evolved network will never have a richer structure (more neurons and/or links) than the template network. There are two main methods you will have to use, namely *addFeature()* (for features see below) and *prepare()*. The latter method must be called after addition of all features to be evolved, as it then determines the precise chromosome structure for network evolution. Please, be aware that netEvo handles all genotype and phenotype settings of JEvolution, so you should not interfere with these settings, i.e., do nothing at all.

The **netEvolver** handles the construction of an evolved network, and most of its methods should not be accessed by the user. The same holds for the **trainEvolver** handling the construction of evolved training parameters and data sets. Hence, the user must only be concerned what he wants to evolve, but not how it is evolved.

The **NetPhenotype** class is the Phenotype implementation of an ANN for JEvolution. Basically, it delegates all the work of decoding the chromosomes to **NetEvolver** and **TrainEvolver,** which are sub-classes of **Evolver**. The only user-relevant method is *getNet()* returning the network representing the phenotype.

The specific aspects of a neural network system, which should be evolved are called *Features* in netEvo and can be found in the **netevo.features.*** package. Only a few examples are listed in the following:

- **Neurons** is the feature for the evolution of neurons, i.e., neurons may be pruned from the template network. Separate switches can be set here for the evolution of input/output neurons.
- **Links** is the feature for the evolution of network connections, i.e., connections may be pruned from the network. Implicitly, this feature may also cause the pruning of neurons, if the pruning of connections lead to a "dangling" neuron being non-functional.
- **Weights** is the feature responsible for the evolution of weights, i.e., the link weights are determined by evolution.
- **Patterns** is a feature supporting the evolution of training data. Evidently, this is only meaningful, if network training is used during evolution. By means of this feature a subset of the given training data set is used for training potentially leading to a more efficient ANN training.
- **Functions** is a feature responsible for the evolution of activation functions. The user has to determine the set of functions to be used, and then evolution assigns specific functions to specific neurons. You may also add a (Boone) spline function, whose parameters are evolved, too, leading to an arbitrary shape of the activation function.

- **Thresholds** is a specific feature for the evolution of Spiking Neural Networks (SNNs). For all the hidden and output neurons the threshold potential, which must be exceeded to cause the firing of a spike, is evolved. In a sense this changes the "sensibility" of a neuron.

The minimal and maximal values of feature parameters are set to default values, but may be changed by the user. Note that in some cases values different from the default values may not be meaningful.

The specific encoding of a feature is handled by a *Decoder* found in the package **netevo.decoders.***. The decoder has to be supplied already for the construction of a feature, e.g.,

*Feature feature = new Weights(new BitDecoder());*

In the latter case a weight evolution feature is constructed using a bit string encoding. Each decoder has a single chromosome associated with it. Thus, using the same decoder for a number of features results in the grouping of features on this single chromosome. If you are using different decoders (objects) for different features, the features will be encoded on different chromosomes. Hence, the user has (nearly) full control over the grouping of features onto chromosomes. Note that the order of features on a specific chromosome is determined by netEvo.

The length of a single parameter on the chromosome is called *geneLength*, which is an attribute of the decoder. Again, this parameter has always a default setting, but the user may change it. E.g., for real-valued parameters encoded with a bit string, the length (number of bits) of the parameter determines the numerical resolution of the parameter.

A **FitnessFunction** computes the fitness of an evolved network, which is usually a number in the unit interval. Larger fitness values indicate better networks. The following fitness functions are available:

- **Classification** – returns the classification accuracy on a pattern set. The returned value is *1 - ( errorCount / patternCount)*, where *errorCount* is the number of misclassified patterns. The class label of a pattern is determined by winner-takes-all, i.e., the output neuron with the largest activation represents the class label assigned to the pattern at the input.
- **NetError** – returns 1 / (1 + SSE) on a pattern set, where SSE is the sum square error over all output neurons and patterns.

A **RegularizationFactor** may be included into the fitness function in order to not only assess the performance of the network but also its complexity. The regularization factor utilizes a regularization weight to modify the fitness as:

*regFitness = fitness * ( 1 – regWeight ) + regFactor * regWeight*

The regularization weight must be chosen carefully (if in doubt use the default value), as a too large weight may drive evolution towards very small networks with low performance.

The following regularization factors are available:

- **REG_NEURON** – number of hidden neurons, *1 / (1 + hiddenNeuronCount)*
- **REG_LINK** – number of links, *1 / (1 + linkCount)*
- **REG_HINTON** – Hinton regularization factor taking into account all the weight values in the network, *1 / (1 + sum(weight^2)*

- **REG_RUMELHART** – Rumelhart regularization factor, also taking into account all the weight values in the network using a slightly different formula, *1 / (1+ sum(weight^2 / (1 + weight^2)))*

# 3. A netEvo application in detail

Below is the source code of an application using the netEvo package. Summarizing the sample program, these are the main steps to be done to evolve a network with netEvo:

1. Construct the **Evolver** with the template network.
2. Add **Features** and **Decoders** telling netEvo what and how to evolve.
3. Call the **Evolver's** *prepare()* method to set up the **JEvolution** genotype and phenotype.
4. Set **JEvolution** parameters and call *doEvolve()*.

The application is a single class named **TestEvolve.** As usual, it starts with the *main()* method:

```
/*
 * Copyright (c) 2014.
 * August Mayer & Helmut A. Mayer
 * All rights reserved.
 */

// imports deleted here

/**
 * A sample netEvo application evolving a network for the letters problem, where 26 different
 * letters are to be classified.
 *
 * @author August Mayer
 * @author Helmut A. Mayer
 */
public class TestEvolve {

    /** The starting point.
     *
     * @param args        arguments (unused)
     * @throws Exception  just pass through potential exceptions
     *
     */
    public static void main(String[] args) throws Exception {

        new TestEvolve().testLetters();
    }
```

The *testLetters()* method contains the actual code. It constructs a template network, reads the pattern file '*letters.pat*', and evolves a network for classification. The Boone class SNNSPatternFile is used to convert the patterns from the SNNS format.

```
    public void testLetters() throws IOException, JEvolutionException {

        /* Create network and load patterns. */
        System.out.println("Creating test network.");
        NeuralNet net = NetFactory.createFeedForward(new int[]{35, 30, 26},
                true,    // fully connected
                null,    // Sigmoid activation function
                null,    // Rprop trainer
                null,    // Standard neuron
                null);   // Standard link

        System.out.println("Loading test patterns.");
        SNNSPatternFile patternFile = new SNNSPatternFile();
        FileInputStream fin = new FileInputStream("test/letters.pat");
        PatternSet pat = patternFile.read(fin);
        fin.close();
```

Next, the **Evolver** is set up for evolution without learning, a **BitDecoder** is constructed (i.e., bit string encoding), and evolution of links, neurons, and weights is set up by the corresponding features. In this case all features are encoded on a single chromosome. If all three features should be on different chromosomes, you simply have to construct a specific decoder (object) for each feature. Some more options are under comments (also in the following).

```
        /* Set up evolver and add features. */
        System.out.println("Setting up features to be evolved.");

        FitnessFunction ff = new Classification(pat);
        Evolver evolver = new Evolver(net, ff);        // net evolution, no learning
//      Evolver evolver = new Evolver(net, pat, ff);   // net evolution with learning

        Decoder decoder = new BitDecoder();            // one decoder for all features =
                                                       // single chromosome
        evolver.addFeature(new Links(decoder));
        Neurons neurons = new Neurons(decoder);
//      neurons.doInputs(true);                        // also evolve input neurons
        evolver.addFeature(neurons);
        evolver.addFeature(new Weights(decoder));
```

Here is just an example, how to change the number of bits used for a parameter, e.g., the number of training epochs.. In this case it is a bit complicated, as the different back-propagation training parameters are encoded in seperate features (internally), so you have to first extract the epochs feature using *getField()*, and then, set the feature's decoder to the desired gene length. Note that this feature is only meaningful, if the **Evolver** is set up for additional training.

After all features have been added, you **must** call the *prepare()* method so as to tell **netEvo** to set up the genotype and phenotype for **JEvolution**. You should not interfere with these settings. E.g., by calling this method the exact position of a feature on a specific chromosome is computed.

```
//      BPTraining bp = new BPTraining(decoder);
//      bp.getField(BPTraining.EPOCHS).getDecoder().setGeneLength(10); // 10 bits for epochs
//      evolver.addFeature(bp);

        /* Sets up genotype and phenotype for evolution. */
        evolver.prepare();
```

Next, general **JEvolution** parameters should be set. Note that the settings below are certainly not the best choices, as they are governed by run time considerations, i.e., the user should not wait minutes and hours for the result of this sample application.

```
        /* Set up JEvolution. */
        System.out.println("Setting up evolution.");
        JEvolution evo = JEvolution.getInstance();
        evo.setPopulationSize(200);
        evo.setMaximalGenerations(100);

        /* Set the mutation rate. This has to be done after prepare(), as the latter
         * determines the length of the chromosome(s). */
        Chromosome c = decoder.getChromosome();
        c.setMutationRate(3.0 / c.getLength());        // 3 is an educated guess
        c.setSoupType(Chromosome.BIASED);              // may improve start generation
```

Now, the set-up is complete, and we may print out all the details, and then, start evolution. Note that all features have the *toString()* method implemented giving some information on the feature and its decoder.

```
        System.out.println(evolver);    // prints the evolution setup

        /* Run JEvolution. */
        evo.doEvolve();
```

During evolution the JEvolutionReporter gives some information on the evolution progress. The amount of information may be set using the **JEvolution** method *setReportLevel()*. After evolution the best evolved network may be saved to a file using the **Boone** method *save()*. Note that only here you have to be concerned with the **NetPhenotype**, which otherwise is only internally used by **netEvo**.

```
        /* Save the best network to a file. */
        NetPhenotype phenotype = (NetPhenotype)
            evo.getJEvolutionReporter().getBestIndividual().getPhenotype();
        NeuralNet bestNet = phenotype.getNet();
        System.out.println("Save best net.");
        bestNet.save(new File("best.xnet"));
    }
}
```

The test application **SpikeApp** focuses on the evolution of SNNs. In addition to all standard ANN features, specific SNN features may be evolved, namely **Thresholds**, **Delays**, **AbsRefPeriod**, and **RelRefPeriod**. Changes of the default minimal and maximal values should be pondered with great care, as inappropriate settings may cause the SNN to not generate any spikes, or even worse, lead to unepredictable behavior of the SNN. The simple task for the SNN in the test application is to (roughly) generate a number of output spikes being equal to the number of input spikes. The input spikes for each evolutionary run are generated randomly with fring times in the interval from 0 to 1 second. The input spke set does not change during a run, so an SNN is adapted to the specific random input set.