

July : Recurrency in joone

Introduction

This is not really an introduction into recurrent networks, nor is it meant to help you get savvy with joone. I write this assuming you have knowledge of both of the above and am able to compile joone on your own, create, configure and train networks from java and so forth.

Joone is a great package and I've used it for years. Initially I cut my teeth on SNNS, which is a great way to start with neural networks. But joone is in java, has a leaner license and is extendable – at least in theory. When I ran into problems with joone, I was able to change the source code or, in trying to do that, learn how to fix the problem.

One problem that I could not address sufficiently was recurrent networks. While joone does cater for these, and has for example a loopback setting for synapses, it does not offer any recurrent network training algorithms. This eventually lead to the development of july, which really is joone with a couple of changes geared towards recurrent networks as well as a few implementations of recurrent learning algorithms.

What this document really tries to do is to help you use july to build such networks. You can and probably should still use the joone gui to build your network, marking the correct synapses as looped back, but then, when you want to train these, july and java code is probably where you should go.

Available algorithms

July has three algorithms, RTRL (real time recurrent learning) and two EKF (extended Kalman filter) ones – the first for normal and the next for recurrent networks. References as well as liberal comments and examples are given in the source code and outlined here. July also contains extensions that allows for the utilization of algorithms from the OAT project to train a network and provides a few new OAT implementations, namely for the Nelder-Mead simplex method as well as for a homegrown systematic differential algorithm that is loosely based on the differential evolution algorithm, which was found to be particularly fast when training networks, but still probably not good enough to train a large, recurrent neural network.

Network structure

The heart and soul of joone is found in the

```
org.joone.structure.NodesAndWeights
```

class. This decomposes a joone network into nodes and a matrix of weights that connect these nodes.

Nodes are classified and ordered and weight references are created so that algorithms can do something like

```
weight.setWeight( value )
```

and joone will figure out in which bias matrix to fiddle to set the weight.

Those who want to develop a whole new algorithm should probably pay some attention to this class.

OAT

OAT (www.sf.net/projects/optalgtoolkit) provides a neat framework in which to optimize real valued problems, such as a neural network. Which set of weights will yield the smallest RMSE for example?

The class

```
org.joone.structure.NetworkAsCFOPProblem
```

can be used to wrap a network inside a CFOPProblem (continuous function optimization) and then throw OAT algorithms at it. The standard OAT algorithms can be used as well as the new

```
com.oat.domains.cfo.algorithms.NelderMead
```

which provides an implementation of the Nelder Mead downhill simplex algorithm or the homegrown

```
com.oat.domains.cfo.algorithms.SystematicDifferential
```

which tries to improve upon the excellent differential evolution algorithm. Examples are given in the

```
org.joone.engine.RTRLearnerFactory
```

class. From this class's testOAT method comes this snippet:

```
// Here we only need number of patterns to use really
network.getMonitor ().setLearners ( null );
network.getMonitor ().setTrainingPatterns ( 0 );
network.getMonitor ().setLearning ( false );

network.getMonitor ().setValidationPatterns ( 1000 );
```

```

network.getMonitor ().setValidation ( true );
network.getMonitor ().setTotCicles ( 1 );

// RTRL, now just used to set up the problem
RTRL LearnerFactory rtrl = new RTRL LearnerFactory ( network, true );

// Prepare an oat stop condition, will end after number of cycles
EvaluationsStopCondition stopCondition = new EvaluationsStopCondition ( 50000 );

// Prepare the oat algorithm and probe to optimise this with
// I always wanted to use Class.forName like this.....
Algorithm algorithm = ( Algorithm ) Class.forName ( args[ i + 1 ] ).newInstance ();
AlgorithmExecutor executor = new AlgorithmExecutor (
    rtrl.getProblem ( args [ i + 1 ], null, 20.0, 0 ), algorithm, stopCondition );
BestSolutionProbe probe = new BestSolutionProbe ();
executor.addRunProbe ( probe );
executor.executeAndWait ();

// Retrieve the best solution of the run and load it into the neural network weights
CFOSolution bestSolution = ( CFOSolution ) probe.getBestSolution ();
for ( int j = 0; j < bestSolution.getCoordinate ().length; j ++ )
{
    rtrl.weights.get ( j ).setWeight ( bestSolution.getCoordinate () [ j ] );
}

// Amuse user
System.err.println ( "Optimal weights loaded" );
rtrl.printWeights ( System.err );

// Run once more to determine error at optimal weights
network.reset ();
network.getMonitor ().setTotCicles ( 1 );
network.go ( true, false );
network.join ();

// Cost is the error at the end
System.err.println ( "Error at this set of weights is " +
    network.getMonitor ().getGlobalError () );

// Also dump to STDOUT
System.out.println ( args [ i + 1 ] + " optimal weights" );
rtrl.printWeights ( System.out );
System.out.println ( args [ i + 1 ] + " optimal error : " +
    network.getMonitor ().getGlobalError () );

// Done, save network
System.err.println ( "\tDone! Saving network" );

```

This snippet shows how to use the RTRL LearnerFactory in order to extract and configure a CFOP Problem that is then passed to one of OAT's CFO Algorithms. Easy! The most difficult thing to use OAT algorithms now becomes setting up the classpath.

The RTRL LearnerFactory has more history and less use at present. It was initially earmarked to wrap the RTRL algorithm using joone's learner factory extension mechanism, but was eventually replaced by the RTRL LearnerPlugin as described below.

RTRL

The RTRL algorithm is contained in the

```
org.joone.engine.RTRL
```

class – this used to be a mega class with a lot of code and comments, but now mostly the comments have survived and the code have been moved into the NodesAndWeights class. This class has references and lots of information as well as a RTRL implementation, with the help of NodesAndWeights of course. The RTRL class is not used directly, rather use the

```
org.joone.engine.RTRL.LearnerPlugin
```

to optimize a network using RTRL. The main and testRTRL methods have examples, from which this snippet comes:

```
// Learner plugin
RTRL.LearnerPlugin rtrl = new RTRL.LearnerPlugin ( true );
rtrl.setNeuralNet ( network );
network.addNeuralNetListener ( rtrl );

// Just cycle the net, the plugin will do the rest
network.getMonitor ().setLearningRate ( 0.00001 ); // Does need a learning rate
network.getMonitor ().setTotCicles ( 1000 );
network.getMonitor ().setLearning ( false );
network.getMonitor ().setTrainingPatterns ( 0 );
network.getMonitor ().setValidation ( true );
network.getMonitor ().setValidationPatterns ( 1000 );

// Now train the network in single thread mode
network.go ( true, false );
network.join ();
rtrl.rtrl.nodesAndWeights.printWeights ( System.err );

// Done, save network
System.err.println ( "\tDone! Saving network" );
```

This snippet can be applied almost as is to a recurrent (or feed forward) network. The only things to change are the parameters such as the learning rate and number of cycles. This of course assumes that the network has been created with the joone gui and that it has its input streams already attached. The

RTRLearnerPlugin takes one boolean argument, which, if true, indicates that the learning rate may be adapted while the network is trained. This is also a home grown attempt at improving results, but seems to work and is worth trying out.

EKF

The extended Kalman filter is implemented following exactly the same approach as above, with the classes

```
org.joone.engine.ExtendedKalmanFilterFFN
```

and

```
org.joone.engine.ExtendedKalmanFilterRNN
```

containing implementations of the EKF FFN and EKF RNN algorithms and the classes

```
org.joone.engine.EKFFFNLearnerPlugin
```

and

```
org.joone.engine.EKFRRNLearnerPlugin
```

wrapping these and intended for use. Both of these wrapper classes contain examples, from which the following snippet comes:

```
// Learner plugin - SMP version
EKFFFNLearnerPlugin ekf = new EKFFFNLearnerPlugin ( 10, 0 );
ekf.setNeuralNet ( network );
network.addNeuralNetListener ( ekf );

// Just cycle the net, the plugin will do the rest
network.getMonitor ().setTotCicles ( 1000 );
network.getMonitor ().setLearning ( false );
network.getMonitor ().setTrainingPatterns ( 0 );
network.getMonitor ().setValidation ( true );
network.getMonitor ().setValidationPatterns ( 1000 );

// Now train the network in single thread mode
network.go ( true, false );
network.join ();
ekf.ekf.nodesAndWeights.printWeights ( System.err );

// Done, save network
System.err.println ( "\tDone! Saving network" );
```

Note how similar this is to the RTRL code and that again the parameters such as total cycles and

training patterns are the only ones that need to be addressed. These can even be set using the gui.

Colt

The EKF implementation's simplicity comes at a price of course – you need the colt library and some more work on the classpath to use these. Colt allows for multiple processors and the EKF caters for this in order to improve performance. EKF takes next to forever to train a decent sized network.

Results and bugs

July is made available in the hope that it will be useful, but without any warranties. The EKF implementations are especially suspect. If you do find any bugs or are able to make improvements, please let me know.