Neural Network learning and regularization as a multiobjective optimization problem

February 5, 2013

1 Introduction

Regularization is a technique commonly used in machine learning to prevent the learners (in our case, neural networks) from overfitting. During training, there is the possibility for a network of overfitting on the training data: the network parameters become adjusted to very specific features of the training data, which bear a minimal or unexistent relationship with the target function. Overfitted networks show good success rates on the training set, whereas they expose extremely low performances on test data. In order to avoid these situations, thus improving generalization capabilities, different kinds of regularization parameters can be employed. They are formally expressed as additional terms for the cost function.

Here, however, a different approach is tried out, described in [1]: the double goal of network learning and regularization is regarded, and thus solved, as a multi-objective evolutionary problem.

2 The experiment

Initially, a population of 100 feedforward neural networks is created. [table 1] shows the initial values chosen for the networks' parameters, or the allowed ranges when picked randomly.
The population undergoes an evolution with a slightly modified version of the multi-objective genetic algorithm NSGA-II; in-between consecutive generations, the networks belonging to the current population are trained with Rprop+. A pseudocode of the whole training process is shown in [1].
See the sections below for a more detailed explanation of each stage.

2.1 The genetic algorithm

NSGA-II ([2]) is employed for commanding the evolution of the population. The objectives considered are the error and the complexity. Both are computed for the current population after the life-long training took place.

Algorithm 1 Pseudocode of the training process

```
trainingSamples, generationTest, finalTest = AcquireSamples();
population = InitFirstPopulation();

for every training epoch:
  train all the networks in the population with Rprop+ using trainingSamples;
  compute the average error of each network for the samples in generationTest;
  apply NSGA-II and generate next population;

networks = paretoFront(population);
test performances of networks using finalTest;
```

Global parameters	
Population size	100
Training epochs	100
Training samples	1000
Population test samples	100
Final test samples	256
Network topology parameters	
Number of hidden neurons	10
Initial weights	[-0.2, 0.2]
Initial biases	[-6.0, 1.0]
Genetic algorithm parameters	
Maximum number of hidden neurons	10
Minimum number of hidden neurons	5
Minimum number of links	15
Standard deviation for Gaussian noise	0.05
Rprop parameters	
η^+	10.2
η^-	0.001
Δ_{max}	50
Δ_{min}	0
Δ_{in}	0.0125

Table 1: Network parameters

The error is the percentage of wrong answers provided for a fixed set of 100 samples, named generationTest in the pseudocode, none of which is used elsewhere during the experiment.
In several empirical studies, the generalization capabilities have been demonstrated to be linked with the network's complexity, or number of links: over-linked networks usually show some form of overfitting. Therefore, this is the second objective considered by NSGA-II, and works as a regularization term.

Both the experiment in the paper and my project don't employ crossover when creating the next population, so some adjustments to the traditional implementation of NSGA-II have been made. Namely, the desired archive size is permanently set to half the population size, in turn never changed. Each network elected to be in the archive breeds a child network through mutation, but it will also be inserted in the next population. No parent selection algorithm, such as tournament selection, is needed, given that all the networks in the archive are "lone" parents.
This, like the classical NSGA-II does, favours a highly elitist approach over random exploration.

The mutation operators are chosen randomly between five possibilities:

- Adding a new hidden neuron;
- Removing a hidden neuron;
- Adding a link between two unconnected neurons;
- Removing a link between two neurons;
- Applying a Gaussian mutation to weights and biases.

It should be noted that the topology limits shown in [table 1] are always satisfied: if a mutation would violate one of those limits, the Gaussian mutation is performed instead.

2.2 Life-long training

The life-long training is performed always on the same training set of 100 samples, and on all the networks in the population before applying the genetic algorithm.
Rprop+, an improvement of the traditional Rprop algorithm described for instance in [3], has being used to train the weights and biases of our networks.

2.3 Conclusions

After all the training is finished, we select the Pareto front of the last population computed by the generic algorithm to be our ensemble. The testing phase then feeds about 250 test samples to the ensemble: each network's answer is regarded as a vote of the single network on the classification of the sample. The ensemble's answer is the class that got more votes.
A simple counting of right and wrong classifications is immediately reported on the command line at each execution, while the last chapter of this report presents more detailed statistics on the overall performances of the experiment.

3 Notes on implementation

3.1 Compiling the program

The project is written in C++, using Qt as a support library. CMake v2.6 or higher is required for building. In order to compile the program execute:

```
mkdir build
cd build
cmake .. && make
./neural
```

Make sure the tictactoe directory is present in the source directory, as it contains the samples used in the project: an error will be issued immediately if it cannot be accessed.

3.2 Network topology

The neural networks are feedforward neural networks with three layers: an input layer, with one neuron for each input attribute needed by the problem; an hidden layer with an initial dimension of 10 neurons; an output layer, with as many neurons as the number of classes.
In the input and hidden layer there are Sigmoid neurons, characterized by the following activation function:

$$output = \frac{1}{1+e^{-z}}$$

$$z = bias + \sum_i weight_i * input_i$$

while in the output layer Tangent neurons are employed:

$$output = \tanh(z)$$

with z defined as above. All the parameters are initially set to random floating point variables withing given ranges. Expecially the range assigned to biases plays an important role, because otherwise we may stumble upon a huge amount of approximation errors in the computation of the output for the sigmoid neurons.

The biases are implemented as weights on fake links with output always set to 1, and a dummy neuron as predecessor: this enables the program to mutate and train them like normal weights. These special links are created separately at the beginning and when a new hidden neuron is added.

In the initialization phase links between neurons are created with probability 0.5, so the networks may not be fully connected. No links between neurons in the same layer are allowed.

3.3 The problem

The problem being solved is a problem of binary classification, taken from `http://archive.ics.uci.edu/ml/datasets.html`.
Please note that various parts of the implementation rely on the output layer having just one neuron. The class the network gave as an answer is discriminated simply by looking at the sign of this neuron's output. If the project must be adapted to different problems, a revision of the implementation is suggested.

3.4 The classes

Here is a brief description of all the classes and modules that compose this project:

- **Neuron**: this abstract class defines the generic properties of any neuron, such as the incoming and outgoing connections, the layer, and an identifier which is unique inside the network. A virtual pure method computeOutput() is declared: it must be defined by all the concrete subclasses, so that it will contain the actual computation needed for the output. This method, once the output is known, must set it to all the outgoing connections, otherwise it won't be accessible to the following layers: please refer to the following non-abstract subclasses for examples.
- **TangentNeuron** and **SigmoidNeuron**: they inherit Neuron and explicit the specific behaviour of the correspondent category of neurons. All neurons can be copied safely using the copy constructors, without losing parameters.
- **Link**: a class for a link between two neurons, referred to as predecessor and successor.
- **LinkMatrix**: a class for holding all the links in a network, indexed by the identifiers of the two neurons connected by it.
- **Network**: a neural network with the characteristics described in the section about topology. The Rprop+ algorithm, complete with computation of the gradients, is implemented here.
- **Ensemble**: holds the network ensemble and performs all the training and testing. The NSGA-II's implementation is thus found in this class.
- **ProblemInfo**: a singleton which reads the problem samples from the hard disk and randomly permutes them. Also most fixed-value parameters used thorought the application are macros in the ProblemInfo.h header.
- **Utils**: a module containing some useful functions for generating random numbers and applying the Gaussian mutation.

4 Empirical results

In this chapter I will present some empirical results of my project, first using the default table of parameters shown in [table 1], and then operating some changes on the parameters to see what happens.

In [figure 1], it shows the results of our network's ensemble after all the training took place, i.e. the percentage of correct answers over a test set of 256 samples, chosen randomly from the samples file. Such results were collected for a total of 20 executions. The final arithmetic average is also shown.

The following plots all show results for one execution. However, they have been carefully picked in order to represent the most average results I got locally.

[figure 2] plots the evolution of the average error of the population during epochs. After the life-long training has taken place, we first compute the average error of each network in the population; then an average over the population size is computed to find the average performance of the entire population.

On a different execution, we plotted the same average error, but with a smaller granularity. Using [figure 3], we notice that although there is still a tendency towards lower values, after an initial steep descent we experiment some oscillations.

As written in the table of parameters, I didn't use the value for $\eta^-$and η^+ suggested in the paper. The reason for this is that I noticed that although the average error tends to stabilize towards the end around the usual values, there are more significant oscillations in between, visible even with a greater granularity in [figure 4].

Changing the range in which initial weights are picked from [-0.2, 0.2] to [-10, 10] predictably has a meaningful influence only on the first epochs: [figure 5]. In fact, this range is used only when creating the first population, to choose initial values for the weights, or when adding a new hidden neuron. After this initial increase, the average error evolves again in the usual pace.

Finally, I changed the standard deviation σ used when Gaussian mutating the weights and biases in the networks, which also influences the maximum amplitude of the mutation, from 0.05 to 10. This increased the amount of oscillations of the average error during all the training, and also gave slightly worse results in the end, depending on how many times this mutation was actually selected: [figure 6].

The last plots show that most effects caused by the mentioned changes are eventually damped after a sufficiently large number of training epochs. Other variations I tried, such as eliminating one kind of mutation or increasing the number of mutations applied on children networks, caused no significant effect.

References

[1] Y. Jin, T. Okabe, B. Sendhoff. "Neural network regularization and ensembling using multi-objective evolutionary algorithms"

[2] A. Pratap, S. Agarwal, T. Meyarivan. "A fast and elitist multiobjective genetic algorithm: NSGA-II"

[3] C. Igel, M. Hüsken. "Improving the Rprop learning algorithm"

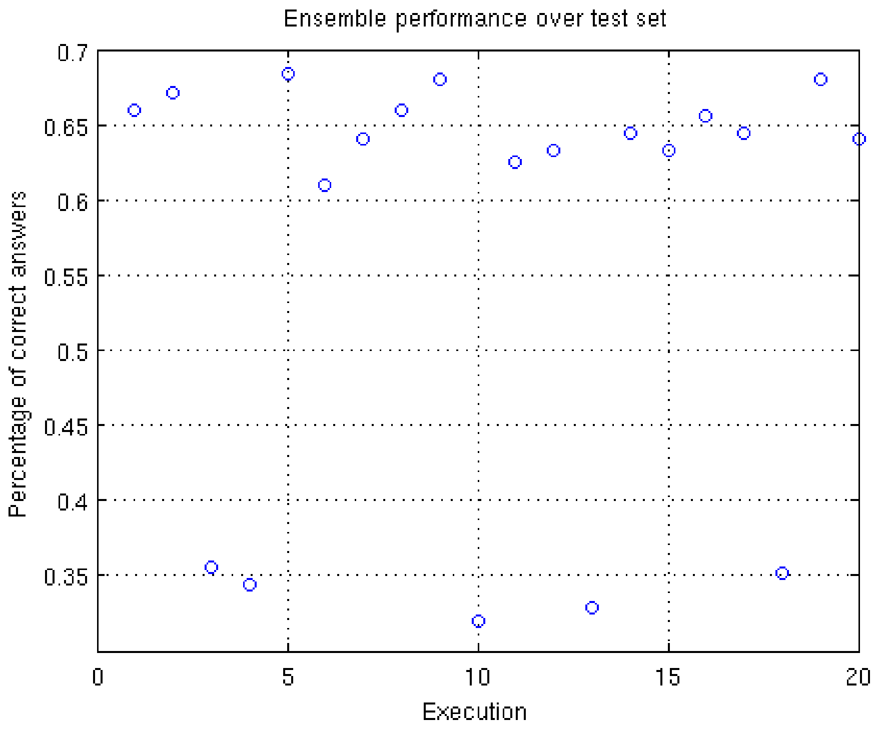

Figure 1: Percentage of correct answers. Average: 57.3%

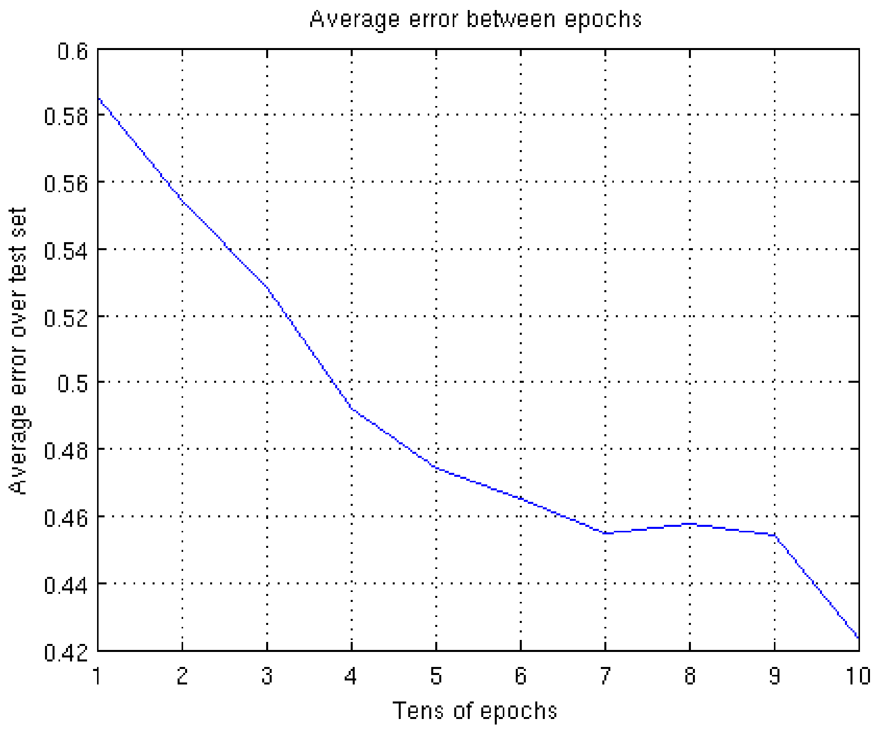

Figure 2: Average error of the population between tens of epochs

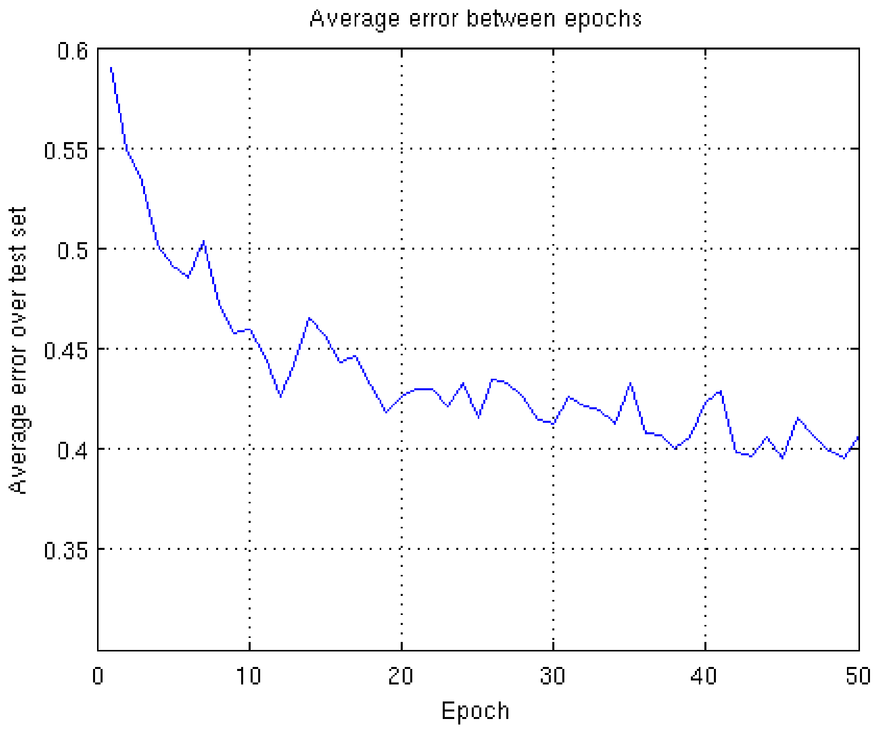

Figure 3: Average error between couples of epochs

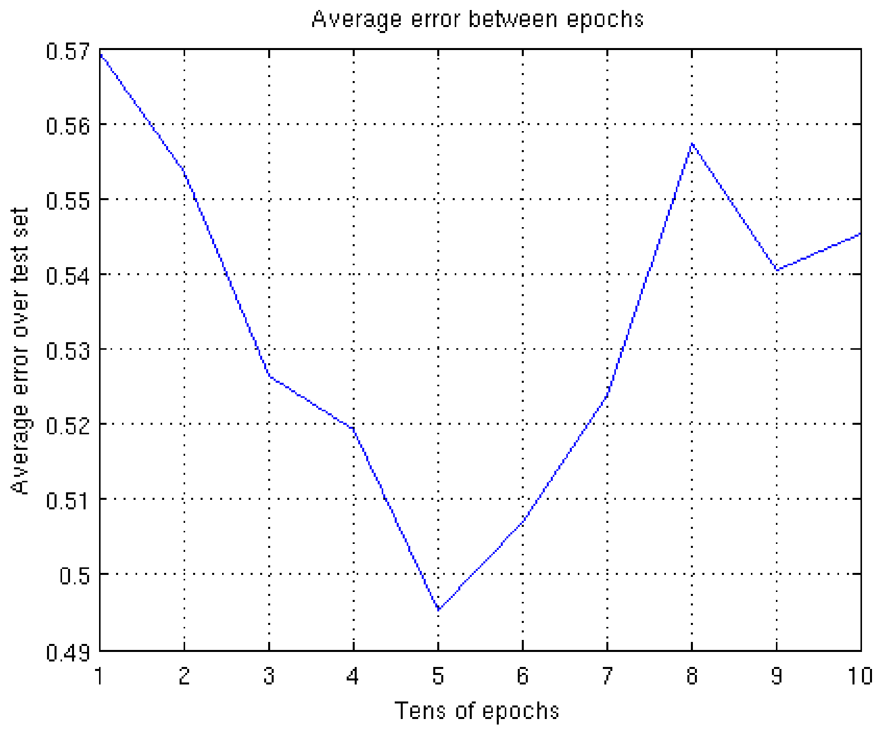

Figure 4: Average error with $\eta^- = 0.2$ and $\eta^+ = 1.2$

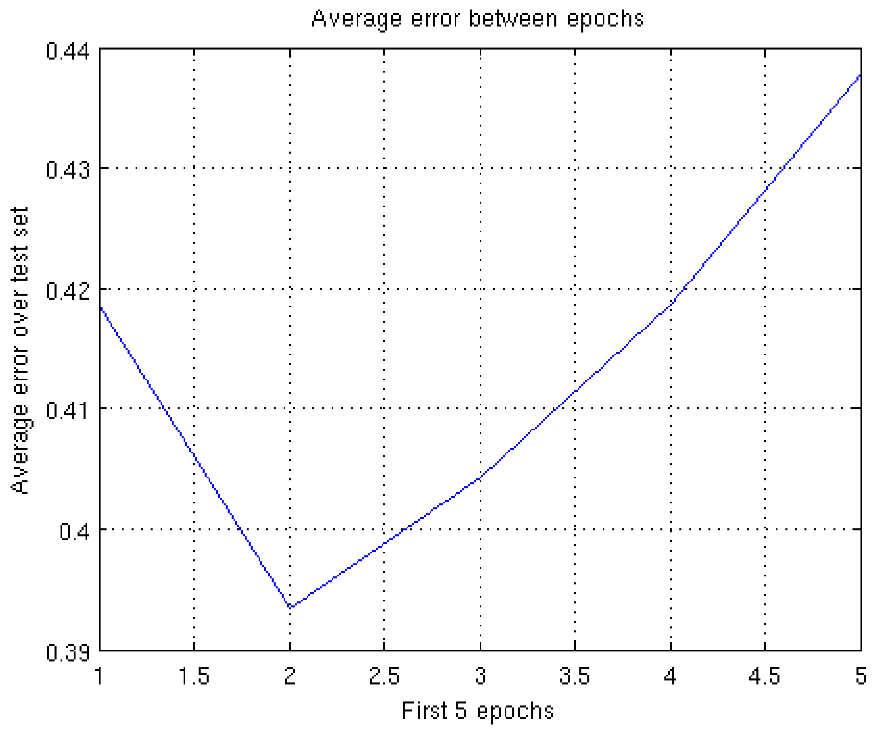

Figure 5: Average error with range [-10, 10] for random initial weights (only first 5 epochs)

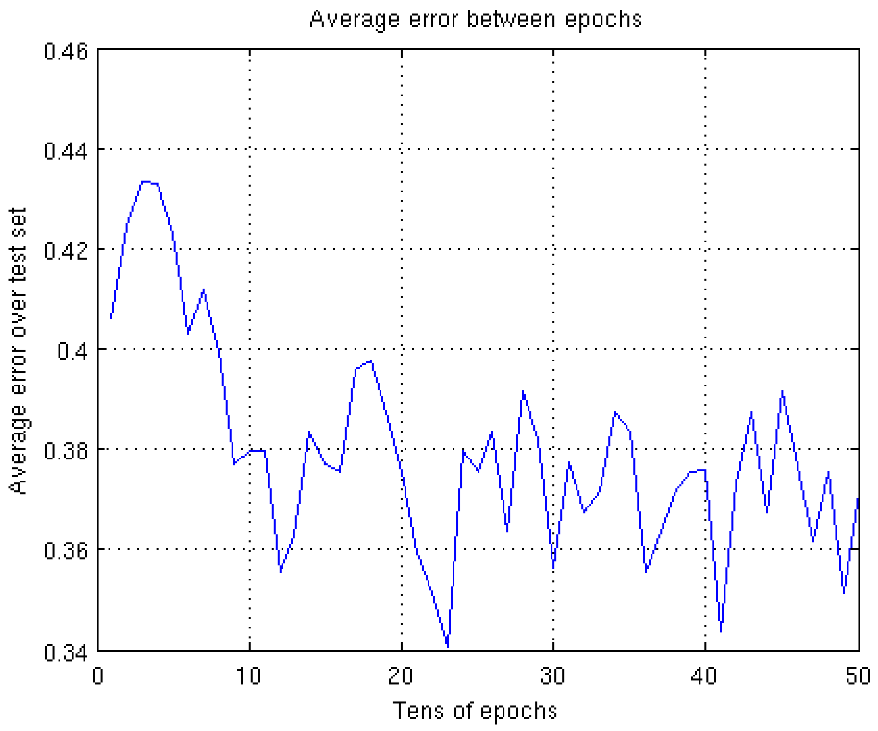

Figure 6: Average error with $\sigma = 10.0$